UFPI – CCN – DC
Ciência da Computação
Estrutura de Dados II

Algoritmos de Ordenação

Prof. Raimundo Moura
rsm@ufpi.edu.br

Algorithms — Robert Sedgewick | Kevin Wayne

2.1 Elementary Sorts

- rules of the game
- selection sort
- insertion sort
- shellsort
- ~~shuffling~~

➢ *Bubblesort*

2.2 Mergesort

2.3 QuickSort

Problema de Ordenação

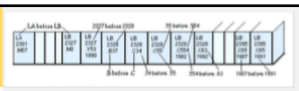
Ex. Student records in a university.

	Chen	3	A	991-878-4944	308 Blair
	Rohde	2	A	232-343-5555	343 Forbes
	Gazsi	4	B	766-093-9873	101 Brown
item →	Furia	1	A	766-093-9873	101 Brown
	Kanaga	3	B	898-122-9643	22 Brown
	Andrews	3	A	664-480-0023	097 Little
key →	Battle	4	C	874-088-1212	121 Whitman

Sort. Rearrange array of N items into ascending order.

	Andrews	3	A	664-480-0023	097 Little
	Battle	4	C	874-088-1212	121 Whitman
	Chen	3	A	991-878-4944	308 Blair
	Furia	1	A	766-093-9873	101 Brown
	Gazsi	4	B	766-093-9873	101 Brown
	Kanaga	3	B	898-122-9643	22 Brown
	Rohde	2	A	232-343-5555	343 Forbes

Aplicações de Ordenação


Library of Congress numbers


FedEx packages


playing cards


contacts


Hogwarts houses

Exemplo de Cliente de Ordenação

Goal. Sort any type of data.
Ex 1. Sort random real numbers in ascending order.

seems artificial (stay tuned for an application)

```
public class Experiment
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Double[] a = new Double[N];
      for (int i = 0; i < N; i++)
         a[i] = StdRandom.uniform();
      Insertion.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

```
% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
```

Exemplo de Cliente de Ordenação

Goal. Sort any type of data.
Ex 2. Sort strings in alphabetical order.

```
public class StringSorter
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readAllStrings();
      Insertion.sort(a);
      for (int i = 0; i < a.length; i++)
         StdOut.println(a[i]);
   }
}
```

```
% more words3.txt
bed bug dad yet zoo ... all bad yes

% java StringSorter < words3.txt
all bad bed bug dad ... yes yet zoo
[suppressing newlines]
```

Exemplo de Cliente de Ordenação

Goal. Sort any type of data.
Ex 3. Sort the files in a given directory by filename.

```
import java.io.File;

public class FileSorter
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles();
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         StdOut.println(files[i].getName());
   }
}
```

```
% java FileSorter .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
```

Ordenação Total

Goal. Sort any type of data (for which sorting is well defined).

A total order is a binary relation $\leq$ that satisfies:

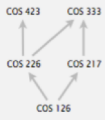
- Antisymmetry: if both $v \leq w$ and $w \leq v$, then $v = w$.
- Transitivity: if both $v \leq w$ and $w \leq x$, then $v \leq x$.
- Totality: either $v \leq w$ or $w \leq v$ or both.

Ex.

- Standard order for natural and real numbers.
- Chronological order for dates or times.
- Alphabetical order for strings.

No transitivity. Rock-paper-scissors.
No totality. PU course prerequisites.

violates transitivity

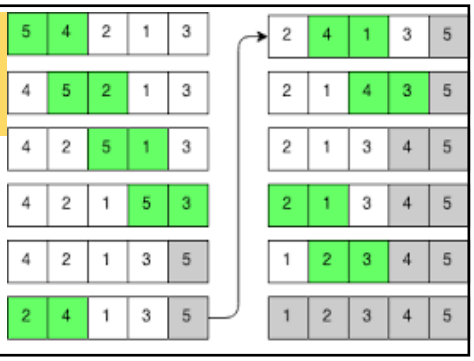


violates totality

Algoritmos Elementares: *Bubblesort*

- Um dos mais simples algoritmos de ordenação (Método da Bolha)
  1) Percorrer o vetor inteiro comparando elementos adjacentes (dois a dois);
  2) Trocar as posições dos elementos se eles estiverem fora de ordem;
  3) Repetir os passos acima com os primeiros n-1 itens, depois com os primeiros n-2 itens, até que reste apenas um item.

Algoritmos Elementares: *Bubblesort*

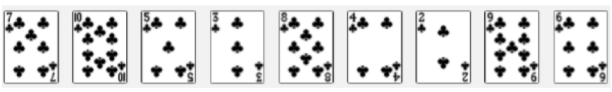


Algoritmos Elementares: *Bubblesort*

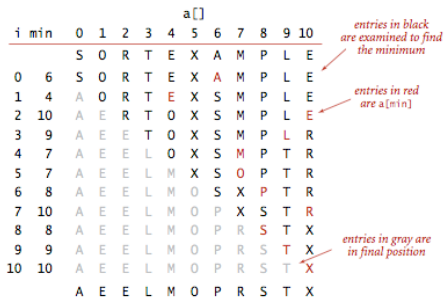
- COMPLEXIDADE:
  - *Pior Caso: $O(n^2)$*
  - *Caso Médio: $O(n^2)$*
  - *Melhor Caso: $O(n)$*

Algoritmos Elementares: *Selection sort*

- Um dos mais simples algoritmos de ordenação
  1) Achar o menor item no array e trocar com a primeira entrada
  2) Encontrar o próximo menor item e trocar com a segunda entrada
  3) E assim sucessivamente



Algoritmos Elementares: *Selection sort*



Trace of selection sort (array contents just after each exchange)

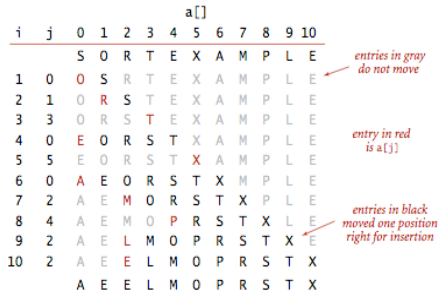
Algoritmos Elementares: *Selection sort*

- PROPOSIÇÃO:
  - Realiza $\sim n^2/2$ comparações e n trocas para ordenar um array de tamanho n.
- COMPLEXIDADE:
  - *Pior Caso: O(n^2)*
  - *Caso Médio: O(n^2)*
  - *Melhor Caso: O(n^2)*

Algoritmos Elementares: *Insertion sort*

- Algoritmo que as pessoas normalmente usam para classificar cartas de baralho. Cada carta é inserida em seu devido lugar entre as cartas já consideradas (mantendo-as ordenadas)
- Antes de inserir o item atual em uma posição, é preciso criar espaço, movendo os itens maiores uma posição à direita.

Algoritmos Elementares: *Insertion sort*



Trace of insertion sort (array contents just after each insertion)

Algoritmos Elementares: *Insertion sort*

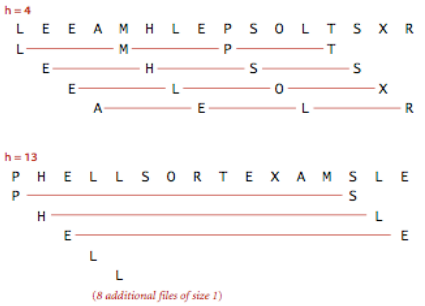
- PROPOSIÇÃO:
  - Para arrays ordenados randomicamente de tamanho n com chaves distinta, realiza $\sim n^2/4$ comparações e $\sim n^2/4$ trocas em média. O pior caso é $\sim n^2/2$ comparações e $\sim n^2/2$ trocas e o melhor caso $n-1$ comparações e 0 trocas.
- COMPLEXIDADE:
  - *Pior Caso: O(n^2)*
  - *Caso Médio: O(n^2)*
  - *Melhor Caso: O(n)*

Algoritmos Elementares: *Shellsort*

- Extensão simples de *Insertion sort* que ganha velocidade ao permitir trocas de entradas que estão muito distantes, para produzir arrays parcialmente ordenados que podem ser ordenados com eficiência, eventualmente com *Insertion sort*.

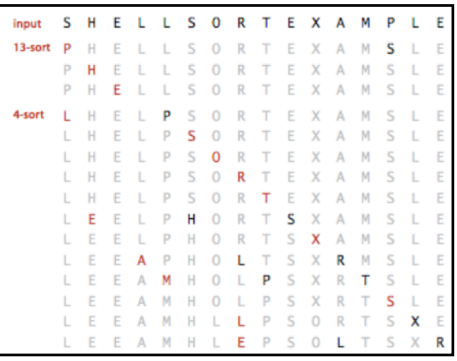
- A ideia é reorganizar o array para fornecer a propriedade que, tomando todas as h-ésimas entradas (iniciando em qualquer lugar), produz uma sequência classificada.

Algoritmos Elementares: *Shellsort*

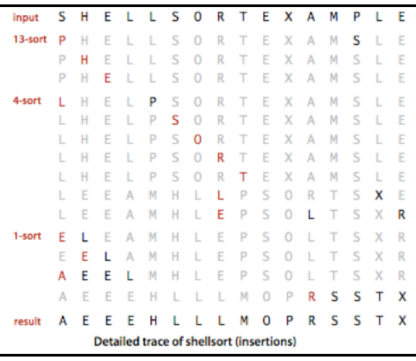


An h-sorted file is h interleaved sorted files

Algoritmos Elementares: *Shellsort*



Algoritmos Elementares: *Shellsort*



Detailed trace of shellsort (insertions)

Algoritmos Elementares: *Shellsort*

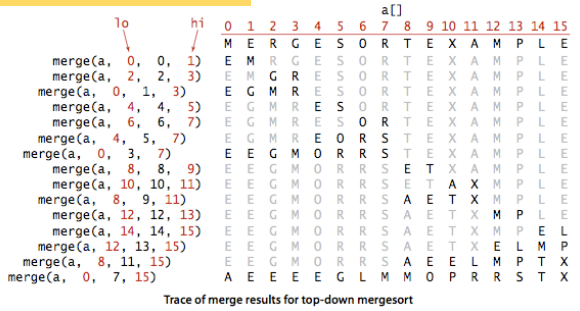
- PROPRIEDADE:
  - O nº de comparações realizadas com incrementos 1, 4, 13, 40, 121, 364, … é limitado por um pequeno múltiplo de *n* vezes o nº de incrementos usados
- PROPOSIÇÃO:
  - O nº de comparações realizadas com os incrementos é $O(n^{3/2})$
- COMPLEXIDADE:
  - *Pior Caso: depende do gap: O(n log n)*
  - *Caso Médio: depende do gap*
  - *Melhor Caso: O(n log n)*

Mergesort

- Baseado na operação simples *"merging"* que combina dois arrays ordenados para fazer um array ordenado maior.
- Para ordenar um array, divide-o em duas metades, ordena-se as duas metades recursivamente e *merge* os resultados

Mergesort overview

Mergesort: Top-Down



Trace of merge results for top-down mergesort

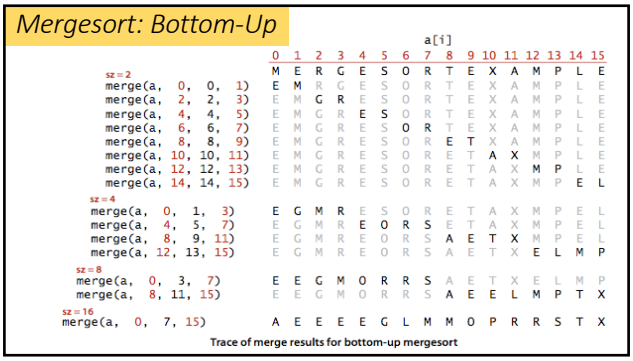
Mergesort: Top-Down

- PROPOSIÇÃO:
  - A versão top-down realiza entre ½ *n log n* e *n log n* comparações e pelo menos *6n log n* acessos para ordenar um array de tamanho N.
- COMPLEXIDADE:
  - *Pior Caso: O(n log n)*
  - *Caso Médio: O(n log n)*
  - *Melhor Caso: O(n log n) ou O(n) para nr. naturais*

Mergesort: Bottom-Up

- Embora estejamos pensando em termos de unir dois grandes subarrays, o fato é que a maioria das fusões está mesclando pequenos subarrays.
- Organizar os *merges* para que façamos todas as mesclagens de pequenos arrays em um passo, depois façamos um segundo passo para mesclar esses arrays em pares e assim por diante, continuando até fazermos uma mesclagem que englobe o todo o array.

Mergesort: Bottom-Up



Trace of merge results for bottom-up mergesort

Mergesort: Bottom-Up

- PROPOSIÇÃO:
  - A versão bottom-up realiza entre ½ *n log n* e *n log n* comparações e pelo menos *6n log n* acessos para ordenar um array de tamanho N.
  - Nenhum algoritmo de ordenação baseado em comparações pode garantir classificar N itens com menos de $\log(n!) \sim n \log n$ comparações
  - *Mergesort* é um algoritmo asintoticamente ótimo

Mergesort: Bottom-Up

- COMPLEXIDADE:
  - *Pior Caso: O(n log n)*
  - *Caso Médio: O(n log n)*
  - *Melhor Caso: O(n log n) ou O(n) para nr. naturais*

Mergesort

- MELHORIAS:
  - Usar *Insertion sort* para pequenos arrays pode melhorar o tempo de execução de 10 a 15%
  - Testar se o array já está ordenado
  - Eliminar a cópia para o array auxiliar

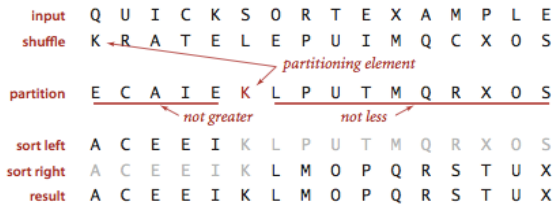
Quicksort

- Popular porque não é difícil implementar
- Trabalha bem para uma variedade de diferentes tipos de dados
- Mais rápido do que qualquer outro método de classificação em aplicações típicas

- Quanto ao espaço, usa apenas uma pequena pilha auxiliar
- Requer tempo proporcional a *n log n* para ordenar N itens

Quicksort: funcionamento básico

- Particiona o array em duas partes e, então, as ordena independentemente



Quicksort overview

Quicksort: Funcionamento básico

- O cerne do método é o particionamento, que rearranja o array para garantir as condições:
  1) A entrada *a[j]* está na sua posição final no array, para algum j;
  2) Nenhuma entrada de *a[lo]* a *a[j-1]* é maior que *a[j]*;
  3) Nenhuma entrada de *a[j+1]* a *a[hi]* é menor que *a[j]*;

OBS: lo é o limite inferior e hi é o limite superior do array

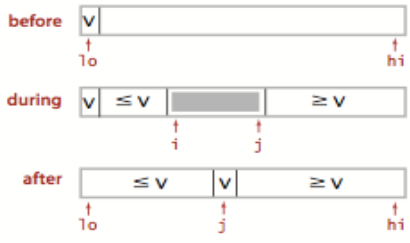
Quicksort: Considerações

- É um *método dividir-e-conquistar*
- Conseguimos uma classificação completa por particionamento, aplicando recursivamente o método aos subarrays.
- É um *algoritmo aleatório*, porque aleatoriamente embaralha o array antes de classificá-lo.

Quicksort: Particionamento

1) Escolhemos *a[lo]* como *pivô* (item que está na sua posição final);
2) Percorremos a partir da extremidade esquerda do array até encontrarmos uma entrada maior que (ou igual a) ao pivô;
3) Percorremos a partir da extremidade direita do array até encontrarmos uma entrada menor que (ou igual a) ao pivô

Quicksort: Particionamento

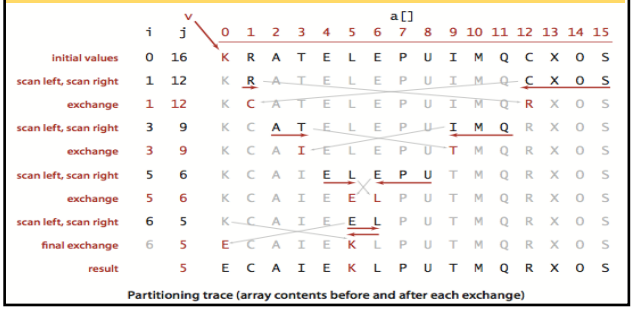


Quicksort partitioning overview

Quicksort: Particionamento

4) Os dois itens (*i* e *j*) estão fora de posição e devem ser trocados;
5) Quando os índices se cruzam, temos que trocar o item *a[lo]* com a entrada mais à direita do subarray da esquerda (*a[j]*) e retornar o índice *j*

Quicksort: Particionamento



Partitioning trace (array contents before and after each exchange)

Quicksort:

Quicksort

- PROPOSIÇÃO:
  - Realiza $2n \log n$ comparações (e $1/6$ que muitas trocas) em média para ordenar um array de tamanho N com chaves distintas.
  - No pior caso faz $\sim n^2/2$ comparações, mas o embaralhamento proteje contra esse caso.
- COMPLEXIDADE:
  - *Pior Caso: O(n²)*
  - *Caso Médio: O(n log n)*
  - *Melhor Caso: O(n log n)*

Atividade Prática

- Considerar uma Tabela de Símbolos com *N* nomes gerados aleatoriamente, implementar três algoritmos de ordenação, sendo que os algoritmos *mergesort* e *quicksort* obrigatórios;
- Realizar experimentos com *N=100, 1000 e 10000*, observando o tempo total gasto para cada procedimento de ordenação;
- Gerar gráficos para cada algoritmo.